Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics
San Antonio, TX, USA - October 2009

An Improved Reduction Algorithm With Deeply Pipelined Operators

Yi-Gang Tai, Chia-Tien Dan Lo, and Kleanthis Psarris
Department of Computer Science
University of Texas at San Antonio
San Antonio, TX, USA
{ytai, danlo, psarris}@cs.utsa.edu

***Abstract*—Many scientific applications involve reduction or accumulation operations on sequential data streams. Examples such as matrix-vector multiplication include multiple inner product operations on different data sets. If the core operator of the reduction is deeply pipelined, which is usually the case, dependencies between the input data cause data hazards in the pipeline and ask for a proper design. In this paper, we propose a modified design of the reduction operation based on Sips and Lin's method. We analyze the performance of the proposed design to prove the correctness of the timing and demonstrate its performance against previous methods.**

***Index Terms*—reduction, algorithm, pipeline, architecture**

I. Introduction

Reduction, or sometimes vector-reduction or accumulation, in the scope of this paper is a process that reduces a set of input data values into a single value. The reduction problem is encountered in designs for many different applications. As an example, in a dot product computation, the results of each multiplicatoin from the two input vectors need to be accumulated. The core operation in the reduction circuit used to reduce the input values can be any commutative and associative binary operator. Without loss of generality, we use the addition operator in lieu of the general binary operator in this paper.

In this paper, we propose a modified design to previous work and analyze the performance of the modified method. The organization of this paper is as follows. In the next section, the reduction problem is defined and the background and previous work are introduced. Following that in Section III, we describe the improved reduction method, the delayed buffering method, proposed in this paper. In Section IV, the performance of the proposed method is analyzed, along with an analytical performance comparison with other methods. Finally, Section V concludes the paper.

II. Background & Previous Work

The reduction problem can be defined in the following. For simplicity, we use the addition operator "+" in place of a general binary operator throughout the paper, and we call it an operation or an addition interchangeably. Let n be the number of consecutive input data elements in a set and X_i be the ith input element of the set and $1 \leq i \leq n$. The reduced scalar output R can be represented as

$$R = X_1 + X_2 + \cdots + X_n. \tag{1}$$

In hardware, this reduction process can be represented as shown in Fig. 1. The binary reduction operation is performed on two input elements at a time, and the partial result is then added by a subsequent input element. While this scheme seems perform well, if the operator is pipelined, an operation needs to wait for its precedent operation to complete before it can enter the operator pipeline. Assume that there are p stages in the operator pipeline, then a new input element has to wait for $p-1$ cycles at the entrance of the pipeline for the previous partial sum to appear at the exit of the pipeline. Therefore, equation (1) is no longer feasible for deeply pipelined case and proper arrangement has to be done to fill the pipeline efficiently to improve the performance.

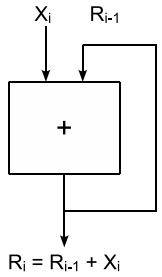

Fig. 1. Reduction in hardware.

The reduction problem has been studied decades ago by Kogge in [1] and Ni and Hwang in [2]. Realizing the binary operator is communicative and associative, the n elements of an input set is partitioned into p groups, where p is the length or latency of the pipeline. The elements in each group are summed independently, and after the reduction in each group is done, the group partial results are then merged to produce the final scalar result. For the Kogge method, the merging is performed using a divide-by-half concept and buffers are needed to store the partial results from the groups. Ni and Hwang take a step further by placing a latch in front of the pipeline entry, eliminating the need of the temporary storage buffers of size p and reducing the computation time.

This work is supported in part by NSF grants EIA-0117255 and CCF-0702527.

In [2], the reduction process is divided into three phases and the reduction time

$$T_r = T_f + T_m + T_d \tag{2}$$

where T_f is the time for the phase to feed all input elements and T_d is the number of cycles for the phase where the pipeline is being drained after the last operation enters the pipeline. It is observed that $T_f + T_d = n + p - 1$ for all the methods studied, and this leaves the term T_m, which is the time spent in the phase to merge all partial results into a single output, the only differing factor among the methods, and the equation for the overall reduction time becomes

$$T_r = n + p - 1 + T_m. \tag{3}$$

Two methods are proposed by Ni and Hwang, namely the symmetric method (SM) and the asymmetric method (AM). These two methods both have similar hardware architecture, but the asymmetric method performs better by recording the state of each stage of the operator pipeline and merging the partial results in a irregular pattern. The merging performance of the asymmetric method in number of cycles is analyzed in [2] as

$$T_m^{AM}(n) = \begin{cases} p\lceil \log p \rceil - 2^{\lceil \log p \rceil} + p & n \geq p \\ p\lceil \log n \rceil - 2^{\lceil \log n \rceil} + n & n < p. \end{cases} \tag{4}$$

Note that in this paper, unless otherwise mentioned, a "log" denotes logarithm operation with a base of 2.

In a correspondence to the Ni-Hwang methods mentioned above, Sips and Lin proposed improved methods for the reduction problem [3]. The improved methods, called modified symmetric method (MS) and modified asymmetric method (MA), overlap the feeding phase and the merging phase and thus improve the performance when the number of input elements is small. The performance, in number of cycles, of the modified asymetric method in the merging phase is

$$T_m^{MA}(n) = \begin{cases} T_m^{AM}(n) - p & n \leq p \\ T_m^{AM}(n) - p + 1 & n = p + 1 \\ T_m^{AM}(\lfloor n/2 \rfloor) - D(\lfloor n/2 \rfloor) & p + 1 < n < 2p \\ T_m^{AM}(n) & n \geq 2p \end{cases} \tag{5}$$

where D is a displacement function that compensates the effect of the irregular merging pattern to the performance. As can be seen in the above equation (5), the modified asymmetric method has significant advantage over the asymmetric method for $n < 2p$. Note that for $n \geq 2p$, the merging performance are the same for both methods. In fact, as will be shown later in this paper, for this type of reduction process the performance is all the same and pipeline is fill if n is large enough.

There are several other approaches in the literature tackling the reduction problem. For some special cases, binary tree accumulators are used for reduction operations, such as the ones in [4] and [5], but the resources required are sometimes not acceptable. In [6], Zhuo et al. proposed an FPGA-based circuit using one adder and $\log n$ fixed-sized buffers for reducing data sets with sizes of a power of 2 but not arbitrary sized sets. Zhuo et al., in [7] and [8], further proposed an one-adder design termed the single strided adder (SSA) method that alternately uses two buffers of size p^2 each and can handle multiple input data sets of arbitrary size. In the design, when one of the buffers is providing data for the reduction process, the other is being used to receive input data elements in the way that it does not overflow. The design carefully and cleverly arranges received data elements into the 2-dimensional $p \times p$ input buffer space so that when the input buffer is swapped to be the reduction buffer, the operator pipeline can be filled without data hazards. These characteristics of the design may not meet the need of some of the possible containing applications, however. Firstly, as has been identified by the authors, it may produce out-of-order reduction results under certain circumstances, yet the pipeline is better utilized. Moreover, for input sets of some sizes, results of different input sets are produced closely in time. For example, if multple sets of size p enter consecutively, after the buffering and the reduction processing, the results are produced at the pipeline output cycle by cycle. This may not be a desired behavior for other circuits using the results of this design. The SSA circuit can reduce m sets of input data in at most $\sum_{i=0}^{m-1} n_i + 2p^2$ cycles, where n_i is the number of elements in set i. Nevertheless, the single set performance of this design is not overly appealing since it focuses on multiple data sets and pipeline utilization.

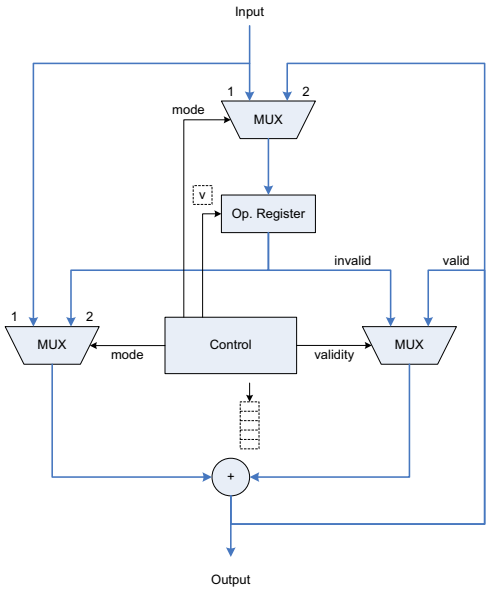

Fig. 2. Hardware configuration for the proposed design.

III. Architecture of the Proposed Method

We discussed in the previous section the modified asymmetric method by Sips and Lin that has better performance than the original asymmetric method when the input set size $n < 2p$. Yet the improvement does not stop there. In this section

Cycle	X	A	B	R	REG
0	X_1				
1	X_2	X_2	X_1		X_1
2	X_3				
3	X_4	X_4	X_3		X_3
4	X_5				
5	X_6	X_6	$X_1 + X_2$	$X_1 + X_2$	X_5
6	X_7	X_7	X_5		X_5
7				$X_3 + X_4$	
8					$X_3 + X_4$
9		$X_3 + X_4$	$X_1 + X_2 + X_6$	$X_1 + X_2 + X_6$	$X_3 + X_4$
10				$X_5 + X_7$	
11					$X_5 + X_7$
12					$X_5 + X_7$
13		$X_5 + X_7$	$\sum_{i=1}^{4} X_i + X_6$	$\sum_{i=1}^{4} X_i + X_6$	$X_5 + X_7$
14					
15					
16					
17				$\sum_{i=1}^{7} X_i$	
18					
19					
20					

(a) $p = 4$

Cycle	X	A	B	R	REG
0	X_1				
1	X_2	X_2	X_1		X_1
2	X_3				
3	X_4	X_4	X_3		X_3
4	X_5				
5	X_6	X_6	X_6		X_5
6	X_7	X_7	$X_1 + X_2$	$X_1 + X_2$	
7				$X_3 + X_4$	
8					$X_3 + X_4$
9					$X_1 + X_2 + X_7$
10		$X_3 + X_4$	$X_5 + X_6$	$X_5 + X_6$	$X_3 + X_4$
11				$X_1 + X_2 + X_7$	
12					$X_1 + X_2 + X_7$
13					$X_1 + X_2 + X_7$
14					$X_1 + X_2 + X_7$
15		$X_1 + X_2 + X_7$	$\sum_{i=3}^{6} X_i$	$\sum_{i=3}^{6} X_i$	$X_1 + X_2 + X_7$
16					
17					
18					
19					
20				$\sum_{i=1}^{7} X_i$	

(b) $p = 5$

Fig. 3. Two illustrative examples of seven input data elements on different pipeline depth for the DB algorithm.

we propose an improved version of the modified algorithm that performs better in certain data set size. Although very similar to the previous methods in the architecture and the resource used, the proposed algorithm is very different from the previous ones conceptually.

A. Hardware Configuration

The proposed hardware configuration is shown in Fig. 2. In this configuration, there are three multiplexors and one operand register besides the operator/adder pipeline; the control logic is shown as a whole in a box in the figure. Different from the one described in [3], the value stored in the operand register can be used as either operands that enter the operator pipeline, but not both at the same time. Furthermore, the input data can only go to the left entry of the pipeline directly. The operand from the right to the operator is always from the feedback path if there is a valid pipeline output.

The three multiplexors are controlled by the mode and validity control signals. The mode signal represents the current processing phase of the circuit. For control purpose, in the proposed design the reduction processing is divided into two phases. When there is currently an input element for the circuit, the circuit is in the loading phase, and obviously this phase occupies n clock cycles. If the reduction unit is not loading new inputs, it is in the joining phase. Control validity signal represents the validity of the current output of the operator pipeline. If at a certain cycle the pipeline produces a valid result, it is always fed back to the right-hand side input of the adder pipeline.

Another control signal, denoted v in the figure, is used to direct whether the operand register should load a new value. In the loading phase, if there is no valid output from the pipeline and the register does not hold any prior input data element, then the current input has to be loaded to the register to keep it from entering the pipeline. In the joining phase, if there is a valid pipeline output but there is no data stored in the register as the other operand for issuing a new operation, the pipeline output is then stored in the operand register.

B. The Algorithm

Conceptually the proposed algorithm works in a very simple way:

- As soon as any two data elements are available, either from the input, the operand register, or the pipeline output feedback path, they enter the pipeline.
- A valid feedback value from the pipeline output always has the highest priority to be consumed by the pipeline.
- If a data element cannot enter the pipeline, it is stored in the operand register.

Although there are two incoming paths, the operand register only needs a capacity of one data element. The reason can be easily observed. From the feedback path, since the feedback pipeline output value is always used first when its available, the only time it is not used there is no value stored in the operand register, and thus it can be stored in the register without overflow. From the input path, since an input element can be paired with either the feedback value or the data stored in the register, similar situation apply when it needs to be stored and there is no problem of storing it in the register. The algorithm is implemented in hardware as what has been shown in Fig. 2 discussed above.

Illustrated in Fig. 3 are two operational examples of how the algorithm works from the aspect of the data present in and out of the pipeline each clock cycle. At clock cycle 0 in both examples, the input data stream starts to appear at the entry of the pipeline. Since it takes two operands for the binary operation, the very first input data element has to be stored in the operand register and wait for the second element. In the next cycle, when the second element enters the reduction circuit, both the element and the first element stored in the operand register will be consumed by the operator pipeline. So far it is exactly the same as the modified asymmetric method presented in [3]. This goes on until the pth cycle where the input element X_{p+1} is present at the entry of the pipeline.

s_1	•	$X_{p+3} + X_{p+1}$	•	$X_{p+3} + C$
s_2	•	$X_{p+2} + (X_1 + X_2)$	•	$X_{p+2} + (X_1 + X_2)$
s_3	○		•	$X_{p+1} + C$
s_4	•	$X_p + X_{p-1}$	•	$X_p + X_{p-1}$
s_5	○		○	
	⋮		⋮	
s_{p-1}	○		○	
s_p	•	$X_3 + X_4$	•	$X_3 + X_4$

(a) DB method (b) MA method

Fig. 4. Difference between the DB method and the MA method at cycle $p + 3$ when p is even.

At clock cycle p, if p is even, there is no value stored in the operand register because all the previous data pairs have been consumed by the pipeline. Hence, the data element X_{p+1} present at the pipeline has to be stored in the register for later usage. In the case of an odd p, because X_p is in the register at cycle p, it can be added by the new element X_{p+1} and both of them enter the pipeline. Here is the major difference between our method and the modified asymmetric method and is clearly depicted in Fig. 4. The $s_1, s_2, \ldots, s_p$ in the figure denote the stages of the operator pipeline. That is, the figure shows the data being handled by different pipeline stages internally. In the MA method, the element X_{p+1} present at cycle p is designed to enter the pipeline paired with the identity element C to prevent collision, while in our proposed method, X_{p+1} is held in the operand register for delayed consumption. In the figure, it can be seen that For this reason, comparing with the MA method, we dub our scheme the delayed buffering (DB) method.

The delayed buffering method may seem slower at first because some data elements enter the operator pipeline later, but as can be seen in the following analysis, it is actually faster than the modified asymmetric method.

IV. Performance Analysis

Although not explicitly established in the algorithm, the approach from equation (2) can still apply to the analysis of our delayed buffering method due to the nature of the problem. $T_f + T_d$ is still $n + p - 1$, which leaves T_m the only differentiating factor for the performance of various algorithms. To analyze the performance of our algorithm, for simplicity, we call a cycle a "hole" for unproductive cycles where not both operands entering the pipeline are valid in the time table such as the one shown in Fig. 3.

It can be observed from time tables that for the DB method, the pattern of holes is exactly the same as the pattern of productive segments at the merging phase of the AM method in [2]. In the beginning, there is no valid operand for the operator pipeline. When input arrives, the unproductive operands enter the pipeline every other cycle, which is also the case at the beginning of the merging phase where the pairs of consecutive pipeline output is fed back to the pipeline every other cycle. Therefore, there are at most $p - 1$ holes and as shown in (4), the last hole appears at cycle $p\lceil \log p \rceil - 2^{\lceil \log p \rceil} + p$, provided that the number of input n is large enough.

Lemma 1. *For n input elements and $k = \lceil \frac{n-1}{p} \rceil$, the number of holes contained in the feeding phase of $n - 1$ cycles equals $\lfloor G(n) \rfloor$, where*

$$G(n) = \frac{n - p(k-1) - 1}{2^k} + \sum_{i=1}^{k-1} \frac{p}{2^i}.$$

Proof: As the pipeline consumes two operands each cycle, the number of holes for the ith iteration of p cycles is $\lfloor p/2^i \rfloor$. The remainding non-integer part $p/2^i - \lfloor p/2^i \rfloor$ affects the location of holes at later iterations. Since it takes 2^i cycles in an iteration to form a hole, the stepping amount for each cycle is $1/2^i$. The remainders accumulate for the formation holes in later iterations. For the last iteration k in which the input of n elements cannot fulfill, the stepping can be represented as $n-1-p(k-1)$ steps over $1/2^k$ each step in the last iteration. The function in the lemma can then be derived. In this sense, the accumulation of previous unrealized holes forms a new hole if G is integer. ■

We can take the $p = 4, n = 7$ case shown in Fig. 3 as an example since $n > p + 1$ in this case. From Lemma 1, $k = 2$ and $G(7) = 2.5$. Thus, there are totally $\lfloor G(7) \rfloor = 2$ holes in the feeding phase, which matches what we can observe in the illustrative example.

Corollary 1. *If $G(n)$ is an integer, then cycle $n - 1$ is a hole.*

From Corollary 1, a cycle can be examined to be a hole or not. However, we need to determine the position of a hole more conveniently. First, from the perspective of performance analysis, we divide the reduction operation into iterations of p clock cycles. Since operands enter the pipeline in pairs, let $q_i = \lceil \frac{p}{2^i} \rceil$ for the ith iteration and in the beginning $q_0 = p$. It can be observed that when q_i is odd, the first possible hole slot in iteration $i + 1$ cannot be eliminated because of the lack of an operand stored in the register for pairing, while in case of even q_i, the first possible hole of the next iteration is occupied. Thus, the accumulated number of holes after the ith iteration is $s_i = \sum_{j=1}^{i} (q_j - E(q_{j-1}))$, where the Boolean function $E(x)$ is equal to 0 if x is even, and 1 if x is odd.

Lemma 2. *The z-th hole for a p-stage pipeline appears at*

$$H_z = kp + (z - s_k)2^{k+1} - \sum_{j=0}^{k-1} 2^{j+1} E(q_j)$$

where $k = i$ if $s_i < z \leq s_{i+1}$ for $i = 0, 1, \ldots, \lceil \log p \rceil - 1$.

Proof: For the first iteration of p cycles, it is obvious that the first k holes are at cycles $2k$ for $k \leq \lfloor p/2 \rfloor$, since it takes a pair of input operands to enter the pipeline productively. It can be observed that later holes are the pipeline results of the holes in the first iteration and their locations are dependent on the previous hole locations. The first hole in the second iteration is located at $H_1 + p$ if $q_0 = p$ is odd, or at $H_2 + p$ if q_0 even. This is also true for the first holes in a later iteration i that depend on q_{i-2}. Moreover, the distance between holes in iteration i is 2^i. Therefore, the location of the z-th hole can

TABLE I
REDUCTION PERFORMANCE COMPARISON OF SEVERAL METHODS IN NUMBER OF CYCLES

	n=5			n=10			n=15			n=25			n=100		
T_r	AM	MA	DB	AM	MA	DB	AM	MA	DB	AM	MA	DB	AM	MA	DB
p=3	12	10	10	17	17	14	22	22	22	32	32	32	107	107	107
p=5	21	16	16	26	26	23	31	31	30	41	41	41	116	116	116
p=8	33	25	25	41	37	35	46	46	43	56	56	55	131	131	131
p=13	53	40	40	68	55	55	76	65	65	87	85	81	161	161	161
p=15	61	46	46	78	63	63	88	73	73	98	95	91	173	173	172

be represented as

$$H_z = \begin{cases} H_{s_{i-2}+1} + p + (z - s_{i-1} - 1)2^i & \text{if } q_{i-2} \text{ odd} \\ H_{s_{i-2}+2} + p + (z - s_{i-1} - 1)2^i & \text{if } q_{i-2} \text{ even} \end{cases} \quad (6)$$

if z is in the ith iteration. By subsituting the H_x in equation (6) from previous iterations, the equation in Lemma 2 can be acquired. Note that when $z = p - 1$, H_z reduces to the form shown in (4). ■

Taking the same $p = 4, n = 7$ example, we demonstrate how to find where the holes are using Lemma 2. Firstly, the accumulated number of holes in the beginning is $s_0 = 0$ and after the first iteration of p cycles is $s_1 = 2$. Thus, for both holes, $k = 0$ in Lemma 2 for this case. Then, using the formula in the lemma, we can find the location of the first hole is $H_1 = 2$ and the second hole is at $H_2 = 4$.

From a time table such as the ones in Fig. 3, we can observe that the number of cycles spent in the merging phase is determined by the time the last pair of operands enter the pipeline. This time is in turn determined by the path that contains the last operation which is also the longest computation path due to operation dependency. If we count the clock cycles bottom-up along the longest path starting from the cycle the last operation enters the pipeline, it can be found that it takes i iterations of p cycles for the last 2^i operations, where the iteration number $i = 0, 1, 2, ...$ Thus, if one of the first operations of the these last ones can be found, the performance of the merging phase of the delayed buffering method can be easily calculated.

To find out where the first of these last 2^i operations is, we start from the cycle of the last input element, counting again bottom-up, and try to find the nearest first operation of longest parth iterations. Since there are $\lfloor G \rfloor$ holes contained in the n input cycles of the feeding phase, with the fact totally $n - 1$ operations are needed for the whole reduction process, there are exactly the same $\lfloor G \rfloor$ operations left in the merging phase. Consequently, we need to count $2^L - \lfloor G \rfloor$ operations bottom-up from the last input cycle but skipping and compensating for the holes because there is no valid operation at those cycles, where

$$L = \begin{cases} \lfloor \log p \rfloor & \text{if } 2^{\lfloor \log p \rfloor} > \lfloor G \rfloor \\ \lceil \log p \rceil & \text{otherwise.} \end{cases}$$

Lemma 3. *The need of compensation for the z-th hole for n input elements can be determined using a compensation check function*

$$C_z = 2^L + H_z - n - z + 1.$$

If $C_z \leq 0$ then no compensation is needed for that hole; otherwise, compensation is needed.

Proof: The construction of the compensation check function is straightforward. Since we need to count $2^L - \lfloor G \rfloor$ productive operations bottom-up from the last input cycle, we check each hole, starting from the last hole, to see if it will be encoutered during the count-up to determine whether compensation is needed. For the last hole, which is the $\lfloor G \rfloor$-th hole located at $H_{\lfloor G \rfloor}$, if the value

$$C_{\lfloor G \rfloor} = \left(2^L - \lfloor G \rfloor\right) - \left(n - 1 - H_{\lfloor G \rfloor}\right).$$

is less than or equal to zero, then the location of the first of the last 2^L operations has been found, and compensation is not needed for the last hole. Otherwise, we have leftovers passing by this hole and still need to count up more. Checking the $(\lfloor G \rfloor - 1)$-th hole can be done in the same way so that

$$C_{\lfloor G \rfloor - 1} = C_{\lfloor G \rfloor} - \left(H_{\lfloor G \rfloor} - H_{\lfloor G \rfloor - 1} - 1\right).$$

By substitution, the check function in the lemma for the kth hole can be obtained. ■

Corollary 2. *The overall compensation for counting the operations bottom-up can be represented as the displacement*

$$D = \sum_{i=1}^{\lfloor G \rfloor} D_i \text{ , where } D_i = \begin{cases} 0 & \text{if } C_i \leq 0 \\ 1 & \text{if } C_i > 0 \end{cases}.$$

From Corollary 2, we know exactly how many more cycles should be counted from the last input cycle up. In the example of $p = 4$ and $n = 7$, we firstly check if compensation for each hole is in need. Using Lemma 3, it can be calculated that $C_1 = -1$ and $C_2 = 0$. Therefore, no count-up compensation is needed for both holes and $D = 0$. The merging performance of the design can now be summarized in the following lemma.

Lemma 4. *For $n > p + 1$, the merging performance for the delayed buffering (DB) method equals*

$$T_m^{DB} = pL - 2^L + \lfloor G \rfloor - D + 1.$$

Proof: The result also comes from a straight derivation. Since it is known that the last 2^L productive operations along the longest computation parth take pL cycles, and the first of these last operations can be found by applying the displacement from Corollary 2, we have

$$T_m^{DB} = \left[n - \left(2^L - \lfloor G \rfloor + D\right) + pL\right] - n + 1$$

and the equation in the lemma can be obtained. Note that if $n \geq pL - 2^L + 2p + 1$, since the input n has passed the last

hole $p+1$ cycles, the pipeline will always be full from then on, and there is no displacement so $D = 0$. That is, $\lfloor G \rfloor - D + 1$ becomes p and T_m^{DB} becomes exactly the same as the case of $n \geq p$ in (4) and the case of $n \geq 2p$ in (5). ■

Theorem 1. *For the delayed buffering (DB) method, the number of cycles spent in the merging phase equals*

$$T_m^{DB} = \begin{cases} p\lceil \log n \rceil - 2^{\lceil \log n \rceil} + n - p & n \leq p \\ p\lceil \log p \rceil - 2^{\lceil \log p \rceil} + 1 & n = p + 1 \\ pL - 2^L + \lfloor G \rfloor - D + 1 & n > p + 1. \end{cases}$$

Proof: Since the DB method behaves exactly the same with the AM and MA methods when $n \leq p + 1$, the performance comes directly from (4) and (5) for n in that range. For $n > p + 1$, the result is from Lemma 4. ■

For the $p = 4, n = 7$ example, since $n > p + 1$, the third case from the Theorem 1 is fit, and we need to know $\lfloor G \rfloor$ and D, which have been previously calculated as 2 and 0, respectively. Thus, for the delayed buffering method, the merging time $T_m = 8 - 4 + 2 - 0 + 1 = 7$, and the total reduction time $T_r = n + p - 1 + T_m = 17$.

The number of cycles required in the reduction process for the various methods mentioned in this paper is shown in Table I for different number of p and n. The DB method performs at least as well as the MA method, while in the range $p + 1 < n \leq pL - 2^L + 2p + 1$, the DB method is in most cases faster than the MA method. Indeed when n is large, the gain becomes less significant; however, in the mid-range of pipline size and input size, the DB method does have some impact. Among the various p and n combinations, the speedup against the MA method can be up to 1.11 (eg., $p = 6, n = 10$). If the DB method is used by an application in this range repeatedly, the clock cycles saved can be considerable.

V. Conclusions

In this paper we proposed an improved version to one of the fastest known reduction algorithm involving deep pipeline. There is minimal addtion in resource requirements to the proposed hardware configuration for the reduction method. The performance of the proposed method is analyzed and proved to have better results than previous methods for certain number of input elements in the range of $p+1 < n \leq pL-2^L+2p+1$. The proposed method can be applied to improve the performance of applictions that have reduction or accumulation computations using deep pipeline.

References

[1] P. M. Kogge, *The Architecture of Pipelined Computers*. McGraw-Hill, 1981.

[2] L. M. Ni and K. Hwang, "Vector-reduction techniques for arithmetic pipelines," *IEEE Trans. Comput.*, vol. 34, no. 5, pp. 404–411, 1985.

[3] H. Sips and H. Lin, "An improved vector-reduction method," *Computers, IEEE Transactions on*, vol. 40, no. 2, pp. 214–217, Feb 1991.

[4] G. R. Morris, L. Zhuo, and V. K. Prasanna, "High-performance FPGA-based general reduction methods," in *Proc. of 10th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2005.

[5] G. R. Morris, V. K. Prasanna, and R. D. Anderson, "An FPGA-based application-specific processor for efficient reduction of multiple variable-length floating-point data sets," in *Proc. of 17th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 323–330.

[6] L. Zhuo, G. R. Morris, and V. K. Prasanna, "Designing scalable FPGA-based reduction circuits using pipelined floating-point cores," in *Proc. of 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 3*. Washington, DC, USA: IEEE Computer Society, 2005, p. 147.1.

[7] L. Zhuo and V. K. Prasanna, "High-performance and area-efficient reduction circuits on FPGAs," in *Proceedings of the 17th International Symposium on Computer Architecture and High Performance Computing*, October 2005.

[8] L. Zhuo, G. R. Morris, and V. K. Prasanna, "High-performance reduction circuits using deeply pipelined operators on FPGAs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 10, pp. 1377–1392, October 2007.